RCP Scenarios

Marc Paterno
CD Special Assignments, FNAL

August 15, 2000

Abstract

The note describes scenarios related to the creation of *RCP* objects. Its goal is to cover all applicable cases, and to serve as the primary design document describing the methods by which *RCP* objects can be introduced into programs. This document is a working document, intended for the developers of the RCP package; it will often have incomplete sections.

This document is appropriate for **RCP** 0.4.

Contents

1 Introduction

The note describes several scenarios related to the creation of *RCP* objects. See the document **Run Control Parameters at DØ** (available in the *docs* subdirectory of the *rcp* package) for a more complete overview of the system. Here, we present brief overview.

In a program, a user gets an *RCP* object by extracting it from an *RCPManager*. This can be done either by specifying an *RCPID* (which one might have obtained from an item in an event), or, more often, by specifying a *package name* and an *RCP name*. These two items, supplemented by a *software release version name* and a *datbase name* are sufficient to specify any parameter set that has been released through the software release procedure. In the future, parameter sets that have been entered into the database through the web interface may not be part of any software release; if they are not, they would have no associated software release version. They would still be tracked by their *own* version tag.

In the scenarios below, we refer to a parameter set that does not include any other parameter sets as a *simple* parameter set. We refer to the human-modifiable text file representation of a parameter set as a *script*. In a script, the containment of one parameter set in another is specified by *embedding*, in which the package name and RCP name indicating the parameter set to be included are specified.

2 RCP Database Objects

An instance of *RCPManager* is connected, upon instantiation, with a specific collection of RCP database objects (instances of concrete subclasses of *AbsRCPDatabase*). The specific classes, and also the specific database pieces, connected to the *RCPManager* are determined by an environment variable `RCP_DATABASE_PATH`. This variable must be set to a colon-delimited list describing each database piece, and the class to be used to communicate with that database piece. For example, the string "official/FileSystemDB : Higgs/FileSystemDB : JUser/FileSystemDB" indicates that three databases are to be used:

1. the "official" RCP database,
2. the Higgs physics group database,
3. the JUser (personal) database. Only this database is writable.

All databases will be connected to using a connected to via a *FileSystemDB* interface. In a future release, there will also be the option of using an Oracle interface.

The *RCPManager* is told which experiment's library release version it is to work with via the environment variable `SRT_BASE_RELEASE`, which is the same variable as used by SoftRelTools. This "version name" is used by the *RCPManager* to create an *RCPName*, as described in Section 5.2.

3 Searching for Scripts

A user may require the ability to override a parameter set stored in a database with one of his own. We support this by providing for the existence of local scripts. When a user requests a parameter set by giving a package name and an RCP name, the system will first look for a script with the appropriate name in the local filesystem. If one is found, this script will be used to create the *RCP* object requested.

The search takes place in a directory tree rooted at the directory specified by the environment variable `RCP_SCRIPT_BASE`. If this environment variable is not defined, then the starting directory is the current working directory, as determined by `getcwd`. The environment variable is expected to evaluate to a directory name of an appropriate format for the environment in which the user is running.

The search expects to see a specific directory structure; this structure is defined to be one which conforms to the SRT standard code directory structure.

The directory structure (starting from the directory *base*) expected by the *FileFinder* is shown in Figure 1. Given a base directory "base", a package name "p", and an RCP name "r", the *FileFinder* will look for a file named `base/p/rcp/r.rcp` on Unix systems. On a Windows NT system, all the forward slashes will be replaced with backslashes, in the standard fashion. In order to support the Unix filesystem, neither the package name nor the RCP name may have embedded spaces. In order to support the Windows NT filesystem, two files in the same directory may not have filenames which differ only in case. Script files which violate any of these requirements, or which are placed in a different directory structure, will not be seen by the *FileFinder*, and will be ignored during the search.

```
base
    package1
        rcp
            rcpname1.rcp
            rcpname2.rcp
    package2
        rcp
            rcpname1.rcp
            rcpname2.rcp
```

Figure 1: The directory structure expected by the RCP system.

4 Specifying parameter sets by *RCPName*

When a user specifies a required parameter set by giving both a package name and object name, the RCP system must supply additional information to create an *RCPName*. The additional information is the following:

1. A *version*; this is the release tag associated with the SRT library release which resulted in the addition of the parameter set to the "official" database piece. For parameter sets in any other database piece, use of this part of the name is left to the discretion of the manager(s) of that database piece.

2. A *database name*; this is the name of the database in which the parameter set resides.

5 Scenarios

In this section, we present several specific scenarios for the creation of *RCP* objects. We organize them (approximately) in order of increasing complexity. In each case, we give a snippet of code (a `void` function `doStuff`), which shows how the scenario begins.

5.1 Given an *RCPID*

5.1.1 Situation

The user requests an *RCP* object by giving an *RCPID* to a local *RCPManager*. The *RCP* object requested is found in one of the databases connected to the *RCPManager*.

```
void doStuff(const edm::THandle<XChunk>& h) {
  edm::RCPID id = h->getRCPID();
  edm::RCPManager* mgr = edm::RCPManager::instance();
  edm::RCP myrcp = r.extract(id);
  // now do stuff with the RCP object
}
```

5.1.2 Result

First, the *RCPManager* is created, as described in Section 2. Since the request is made by using an *RCPID*, there is no searching of the local filesystem for a script.

The *RCPManager* queries each of its *RCPDatabaseServices* objects in order, requesting an *RCPValue* object associated with the given *RCPID*. In turn, each *RCPDatabaseServices* object queries its associated *AbsRCPDatabase* object, requesting an *RCPValue* object associated with the given *RCPID*. The *RCPManager* stops when the an *RCPDatabaseServices* returns an *RCPValue*.

The *RCPManager* then wraps the *RCPValue* in an *RCP* object, and returns it to the user.

If the *RCPManager* had searched through its entire collection of *RCPDatabaseServices* objects, and none had responded positively to the request for an object associated with the given *RCPID*, the *RCPManager* would throw an *XRCPNotFound* exception.

5.2 Given package and RCP names: no local scripts

5.2.1 Situation

The user requests an *RCP* object by giving two strings. The first is the name of a CVS package, and the second is the name of a parameter set associated with that package. The additional bit of information required to uniquely identify the required parameter set is the release version of the DØ library the user is using; this is known by the *RCPManager*, and is determined at the time the *RCPManager* is created, by reading the environment variable `SRT_BASE_RELEASE`.

```
void doStuff(const edm::THandle<XChunk>& h) {
   edm::RCPID id = h->getRCPID();
   edm::RCPManager* mgr = edm::RCPManager::instance();
   edm::RCP myrcp = r.extract("cps_geometry",
                              "CpsChannelGeometry");
   // now do stuff with the RCP object
}
```

No scripts are present in the local filesystem.

5.2.2 Result

The *RCPManager* builds an *RCPName* object from the given strings and the string indicating the experiment's software release version (its own data member). The *RCPManager* then queries its *FileFinder*, to determine whether or not a local script exists. The *FileFinder* indicates that none is found.[1]

The *RCPManager* then queries each of its *RCPDatabaseServices* objects in order, requesting an *RCPValue* object associated with the given *RCPName*. The *RCPDatabaseServices* returnsThe *RCPManager* stops when the an *RCPDatabaseServices* returns an *RCPValue*.

The *RCPManager* then wraps the *RCPValue* in an *RCP* object, and returns it to the user.

If the *RCPManager* had searched through its entire collection of *RCPDatabaseServices* objects, and none had responded positively to the request for an object associated with the given *RCPName*, the *RCPManager* would throw an *XRCPNotFound* exception.

[1]See Section 3 for a description of the *FileFinder* class.

5.3 Given package and RCP names: local script, identical parameter set in "official" database

5.3.1 Situation

The user requests an *RCP* object by giving two strings. The first is the name of a CVS package, and the second is the name of a parameter set associated with that package. See Section 5.2.1 for how the `SRT_BASE_RELEASE` environment variable is used to complete the specification of the *RCPName*. The parameter set specified in this file is identical to one found in the "official" database. The `RCP_DATABASE_PATH` environment variable indicates that no writing to any database should be attempted. The code in this case is identical to that in Section 5.2.1.

A script is present in the local filesystem. The environment variable `RCP_-SCRIPT_BASE` points to the directory `mywork`, which contains a directory `cps_geometry`, which in turn conatins a directory `rcp`. In this directory is a file named `CpsChannelGeometry.rcp`. The environment variable `RCP_DATABASE_PATH` is set to look only in the "official" database.

5.3.2 Result

The *RCPManager* builds an *RCPName* object from the given strings and the string indicating the SRT release version (its own data member). The *RCPManager* then queries its *FileFinder* object, to determine whether or not a local script exists. The *FileFinder* finds the script file, and gives it to the *Parser*. The *Parser* creates an *Script* object, from which the *RCPManager* then creates an incomplete *RCPValue* object. The *RCPValue*object is incomplete because it does not have an *RCPID*, and because it has an incomplete *RCPName*. The *RCPName* is incomplete because it is missing its database name.

The *RCPManager* then queries each of its *RCPDatabaseServices* objects in order, requesting an *RCPValue* object with a parameter set equal to the given one. In turn, each *RCPDatabaseServices* object queries the *RCPValue* object for its *RCPHashKey*; it then queries its associated *AbsRCPDatabase* object, requesting all *RCPValue* objects associated with the given *RCPHashKey*. The *RCPManager* stops when the an *RCPDatabaseServices* returns a complete *RCPValue*.

The *RCPManager* then wraps the *RCPValue* in an *RCP* object, and returns it to the user.

If the *RCPManager* had searched through its entire collection of *RCPDatabaseServices* objects, and none had responded positively to the request for an object with a parameter set the same as that specified in the script, the *RCPManager* would throw an *XRCPNotFound* exception – because no writable database was specified.

If the parameter set found in any of the readonly databases had no matching name, then an exception would be thrown.

5.4 Given package and RCP names: local script, no matching parameter set in any datbase, access to writable database

5.4.1 Situation

Just like it says in the title.

5.4.2 Result

Find script. Make *RCPValue*, with no *RCPID* and incomplete *RCPName*; the *RCPName* is missing a version and database name. Give *RCPValue* to *RCP-DatabaseServices*. *RCPDatabaseServices* extracts *RCPHashKey*, and gets all matching *RCPValue* objects from *AbsRCPDatabase*. Check each of these hits for equality with given *RCPValue*; none match. *RCPDatabaseServices* then calls `put` on its *AbsRCPDatabase*, giving it the *RCPValue*. *AbsRCPDatabase* then issues a new unique *RCPID*, completes the *RCPName* by adding its database name and a new unique version (must be unique within this database piece, for this package name and object name). The version number could just be the ordinal of the new parameter set within this package name/object name pair. Completed *RCPValue* is then wrapped in *RCP* and returned to user.

If there were no access to a writable database, then an exception would be thrown.

5.5 Bootstrap scenario: populating the "official" database from a new library release

5.5.1 Situation

The release managers want to insert the collection of parameter sets for a new library release into the "official" database, with a given tag – the tag is the version of the current release.

5.5.2 Result

The `updatedb` program is used to perform this feat. Before running, the environment variables are set up to allow the process to talk to the "official" database in *writable* mode. `updatedb` then is used to read a list of RCP scripts, and to update the "official" database. It does this by making normal use of an *RCPManager* object, using its `extract` method on each script to be updated in the database.

5.6 Given package and object name, no script, no access to “official” database; read access to “higgs” database

5.6.1 Situation

5.6.2 Result

No script found by *FileFinder*. *RCPManager* asks *RCPDatabaseServices* for all *RCPValue* objects matching the incomplete (missing a version) *RCPName*. *RCPDatabaseServices* responds with a collection of *RCPValue* objects. *RCP-Manager* then selects the one with the most recent timestamp. Wrap and deliver.

The only way to control which version is returned is by setting the value of the environment variable which is named in the file `RCP_DB_NAMES_FILE` to the version you want to retrieve; this only works if the “higgs” database has been managed with controlled versions. If this database has been allowed to generate version tags on its own, then the only possibility is to retrieve the most recent version. Maybe this is what will convince users they want the option to specify the version in `RCPManager::extract()`. Or, maybe not.

5.7 Given package, object name, and version name; local script; same parameter set in writeable database; this is a new version

5.7.1 Situation

Note that this is a completely hypothetical case, because the member function of RCPManager that allowed the user to request a specific version of an RCP object has been removed by request of DØ.

The user requests an *RCP* object by giving three strings. The first is the name of a package, the second is the name of a parameter set associated with that package, the third is its version string.

Because an explicit version string has been specified, the *RCPManager* does not attempt to find a local script. Therefore, the existence of the local script is irrelevant. This scenario then becomes identical to that of Section 5.2.

5.7.2 Result

See Section 5.2. Because the package name, object name, and version have all been specified, but no such name is known to any database, the system is unable to find a match – and an exception is thrown.

If the user wants to find the local script, then he must not specify a version. In that case, we would have the same scenario as specified in either Section 5.3 or Section 5.4.

6 Random Questions I Can't Yet Answer

- How does any scenario involving interaction with the "official" RCP database through the web go? We still don't have requirements in hand for web interaction.
- What authority issues database names and database IDs? The current working assumption is that all IDs for a given experiment will be held in a master database at Fermilab.
- Need to specify a scenario that shows how a new name gets associated with a parameter set. A few cases:

    1. Release system is updating the official database piece. A new version name is being assigned to specific parameter set.
    2. Release system is updating the official database piece. An identical parameter set is being given both a new package name and new object name – two different developers happened, by accident, to create a script containing identical name/value pairs.
    3. User is updating his personal database, with version numbers being issued by the database. A script is given a new package and/or object name, but contains the same parameter set as specified under another name in the database piece.